The Graph Traversal Machine: Close Encounters of the Fourth Kind[*]

Marko A. Rodriguez
Captain, S/V Red Herring
Director of Engineering, DataStax Inc.
Project Management Committee, Apache TinkerPop

Stephen Mallette
Senior Engineer, DataStax Inc.
Project Chair, Apache TinkerPop

(Dated: October 31, 2018)

Apache TinkerPop$^{\text{TM}}$ is an open source graph computing framework. The project is approaching its 10th year of existence and the team is about to embark on its fourth and final major release – TinkerPop4. This article reviews the technical and mythological aspects of The TinkerPop all the while demonstrating how its lore has driven many of its architectural decisions and vice versa. This historical synopsis ends at the project's denouement with a concluding prospectus detailing the major threads of the final version that will enable the integration of any data query language, any data storage system, and any data processing engine. At the point of unification, in the closing chapter, Gremlin will come to understand the meaning of The TinkerPop and that moment will write the final line of code and sentence on a deep rooted project in the field of graph computing.

I. INTRODUCTION

Over the last decade, the Apache TinkerPop$^{\text{TM}}$ team, in its incarnation prior to Apache® and now as a top-level Apache project, has focused on the design and development of a system-agnostic framework for graph storage and processing. TinkerPop's graph data structure is the property graph $G = (V, E, \lambda)$, where a set of vertices V are connected via directed, labeled, binary edges $E \subseteq V \times \Sigma^* \times V$ and both vertices and edges can have an arbitrary number of key/value-properties associated with them such that $\lambda : (V \cup E) \times \Sigma^* \to (\Sigma^* \cup \mathbb{R} \cup \ldots \cup \mathbb{N})$. TinkerPop's graph processing component is a Turing Complete [29] virtual machine with respective bytecode and data flow language known as Gremlin. This graph traversal machine is abstractly defined as

$$G \longleftarrow \quad t \in T \quad \longrightarrow \Psi,$$

where the graph/data G is processed by a set of traversers/processors T according to a traversal/program Ψ. Every traverser has two primary projections: one into the graph G and one into the traversal Ψ, where the steps/instructions in Ψ dictate the next function that traverser t will execute in order to alter its location in G. When the traversers have no more steps in Ψ to execute, they halt and the aggregate of the final G-locations of all traversers in T is the result set of the query [18].

TinkerPop specifies the interfaces for interpreting the information inside the underlying data storage system as a property graph, where, ultimately, any TinkerPop-enabled system has the same "look and feel" as any other TinkerPop-enabled system. Differences between systems lie in their respective time and space tradeoffs which are typically realized as random access vs. linear scans, support for index lookups, in-memory subgraph caching, vertex-centric indices, deductively generated edges, edge compression, and other techniques commonly used in the field of graph computing. Once the underlying data is exposed as the graph G, TinkerPop's graph traversal machine's traversers T traverse the graph according to the constraints defined by Ψ.

The aforementioned architecture has come from a long line of choices made in both TinkerPop's technical specification and corresponding mythological narrative. The TinkerPop lore has been recorded in the project's documentation, tutorials, artwork, blog posts, and academic articles over the years. The sum total of this archive is the mythology of The TinkerPop and this story has enlivened the project with an ethos that reaches beyond the rote, utilitarian approach espoused by modern technologists. So much so that the storyline has served as a guiding inspiration to the technology's direction and as such, is driving TinkerPop into its final act which is to take place in the next major release as TinkerPop4. Prior to describing where TinkerPop4 is headed, it is important to first understand where TinkerPop has come and what aspects of its mythos are determining its last phase of life within the ever burgeoning graph computing space.

II. HISTORY

TinkerPop has gone through 3 major releases and 50 minor releases. The technical design choices made for

[*]Rodriguez, M.A., Mallette, S., "The Graph Traversal Machine: Close Encounters of the Fourth Kind," S/V Red Herring's Ship's Log: Chronicles in the Sea of Cortez, pages 1–10, La Paz, Baja California Sur, Mexíco, October 2018.

TinkerPop have found their justification in their juxtaposition to the continuously evolving narrative of The TinkerPop. The TinkerPop story is developed using non-linear, anecdotal, intuitive, right-brain techiques while the technical aspects of the project are driven by an analytical, rational, left-brained approach [7]. Leveraging the power of myth and science has proved successful in overcoming both technical and plot/character development hurdles over the years and will continue to serve as the modus operandi for TinkerPop's final phase of development. The remaining subsections will discuss the relationship between story and design in the major releases through to TinkerPop3.

A. TinkerPop1: The Language

The beginning of the story, at TinkerPop1, introduces a character named Gremlin who is on a life quest to understand something he calls "The TinkerPop." The concept of the The TinkerPop borrows many themes from the Indo-Aryan philosophies as being that which upholds reality in a Brāhmanic-sense and yet at the same time is ungraspable by psychologically and physically localized beings [12]. Gremlin is not deterred from the seeming insurmountable objective of "coming to terms" with The TinkerPop and reasons that in order accomplish his goal he must search the graph in a manner analogous to a world explorer. This life choice gives rise to a series of epic tales reminiscent of the heroic era [4, 10] with Gremlin fighting to solve numerous domain-specific graph use cases that have been developed over the years.

From a technical standpoint, the analog to Gremlin's wanderings about the world/graph required the development of a data query language that imparts a sense of movement on the user. What emerged was a language for specifying a sequence of steps that dictate the legal paths that a swarm of traversers can take through a graph. Gremlin's fluid, path-oriented approach to data analysis was inspired by the sprawling, flow-like nature of spreading activation potentials within neurological graphs [13, 26]. This style is in contrast to the typical select-filter-format model found in most modern set-theoretic query languages.

The Gremlin language is a functional, concatenative language, where steps are appended to a traversal Ψ and the executing traversers T constrain their movement through the graph G according to those steps. If a is a traversal step, b a traversal, and c a constant, then $\Psi \in a(c^*b^*)^*$, where c could be defined as a traversal that yields a single constant value and thus $\Psi \in a(b^*)^*$. Every Ψ-step $a(\ldots)$ generates a function $f : T^* \rightarrow T^*$, where the ultimate graph computation executed is a functional mapping from T to T' and has the form $(f \circ g \circ \ldots \circ h) : T^* \rightarrow T^*$. Every Ψ-projection of $t \in T'$ is $\emptyset$ as the traversers are halted in G and their final locations are the result set. Ψ is a linear, nested structure. A theoretical implication is that Ψ-traversals are, in fact, graph-structures themselves whose embedding in G [15] will later be used in a proof of the Turing Complete nature of Gremlin and will also play an influential role in bridging the mythological gap between TinkerPop3 and TinkerPop4.

B. TinkerPop2: The Compiler

Gremlin traveled the graph far and wide seeking problem ridden domains that he could elegantly solve with his graph traversal techniques. Each new problem space was a whole new world unto itself with new concepts and things (vertices), new relationships (edges), and new traversals (processes). This patchwork of isolated subgraphs at varying levels of resolution, scale, and import made Gremlin think that

> the unusual aspect of [The] TinkerPop is that every possible "thing" is related to every possible "thing" in every conceivable way possible. [...] Even purely conceptual things enjoy an arbitrary existence: left is to the right of east and up is both beside and within down. [17]

Any vertex can be connected to any other vertex. The reason why a vertex or edge existed or not was simply a matter of the constraints of the builders of that particular world. This disheartened Gremlin. It appeared that there would always be more problems, more graph, and more traversals. He could continue his travels, but his life mission was to understand The TinkerPop and solving the world's concocted worries no longer seemed a means of getting him closer to his goal. If The TinkerPop truly enabled any arbitrary structure, then he was going to create a world (and subsequent problems) all his own. This was a use case that he could care about as it was *his* use case.

English folklore speaks of gremlins as little "machine elves" competent at building (or, more nefariously, deconstructing) physical machines [5]. Gremlin uses his powers of creation to construct a collection of mechanical friends whom he names Blueprints, Pipes, Frames, Furnace, and Rexster. Blueprints was the architect able to mutate the graph. Pipes was the plumber able to manipulate a flow of vertices and edges. Frames was the interior designer able to express complex graph constructs in simpler terms. Furnace was the boiler that moved traversers about the graph. And finally, there was Rexster who was a Gremlin's best friend, happily greeting those who showed up at Gremlin's door. Gremlin had a made a nice home for himself within The TinkerPop. He had a little plot of graph that he could call his own.

From a technical standpoint, TinkerPop2 primarily focused on the advancement of the tooling around Gremlin (the language) with particular focus on Blueprints and Rexster. Blueprints was refactored to account for the growing need from providers to be able to specify more complex data selection patterns. This was inspired by the use of vertex-centric indices in the industry, where a query of the form `outE('knows').has('stars',gt(3))` would best be expressed as a single pushdown-predicate as opposed to a broad query (`outE(...)`) followed by an in-memory filter (`has(...)`). TinkerPop2's direction spawned the first language optimizer for Gremlin that would later advance leaps and bounds in TinkerPop3 and will reach a definitive design in TinkerPop4. Furthermore, TinkerPop was starting to be recognized by other language communities and it was becoming increasingly necessary to evolve the project beyond its original JVM confines. A language agnostic binary protocol and a REST/HTTP protocol were developed for Rexster. This work would later turn into GremlinServer in TinkerPop3 and will serve as the binary protocol and serialization infrastructure for TinkerPop4.

In general, TinkerPop2 was settling down into a stable project that would evolve slowly over time. However, this peaceful repose would not hold for long. Gremlin was becoming restless with his simple life and it was only a matter of time before he would embark on a new effort of blockbuster proportions: TinkerPop3.

C. TinkerPop3: The Bytecode

TinkerPop2 was a refactoring of TinkerPop1. TinkerPop3 was a complete rewrite of the codebase. Rewriting the codebase from scratch helped rectify a number of architectural problems that had started to hinder the progress of the project. Accordingly, the rewrite forced The TinkerPop story to evolve to account for the new design choices and features which were summarily captured in [25] at the outset of the development process. TinkerPop3's major advancements included:

- **A hosted language capability**: The nested, concatenative nature of the Gremlin language enables its embedding within any programming language, where traversals are constructed using method chaining. Nearly every major programming language has a respective Gremlin language variant. [Frames]
- **A bytecode specification**: The graph traversal machine processes a list of bytecode instructions of the form $[op, arg^*]^*$, where any arg can be yet another list of bytecode instructions. Any query language can compile to bytecode in order to execute on the Turing Complete traversal machine and therefore, against any TinkerPop-enabled data system. [Frames]
- **A general purpose compiler**: The traversal machine maintains an ordered list of traversal strategies that rewrite a traversal into an algebraically equivalent form, where the purpose is to optimize the traversal's time/space-requirements with respects to the underlying data storage system [21, 22]. [Pipes]
- **A declarative runtime optimizer**: The Gremlin language was extended with declarative constructs such as `match`-step which borrows its semantics from the graph pattern matching language SPARQL [11]. Furthermore, a runtime optimizer was included that dynamically resorts patterns based on realtime performance statistics [1]. [Pipes]
- **A traversal-centric processor**: Every traversal is responsible for its underlying swarm of traversers. Besides side-effect steps, traversal steps and, in turn, traversals are purely functional in nature as static descriptions of process. Both serial (standard – OLTP) and a distributed (graph computer – OLAP) traversal processors were developed. [Furnace]
- **An objectified traverser**: Traversers were explicitly defined in order to offload state from traversal steps (e.g. path, bulk, loop, etc. data). Traverser atomicity allowed traversers to split and merge according to respective forks and joins in the underlying graph structure. This enabled the development of the *bulking optimization* which greatly increased the speed and decreased the memory footprint of a traversal [18]. [Furnace]
- **A universal serialization format**: Serialization of bytecode to and results sets from the graph traversal machine was made possible via a JSON-based, language agnostic serialization format called GraphSON. For inter-JVM communication, a byte-based serialization format called Gryo was developed. [Rexster]
- **An extension to the property graph model**: In order to meet the requirements of use cases such as auditing, permissions, and security, it became important to be able to express multiple properties for a single vertex key and furthermore, for each vertex property to be able to maintain properties themselves. This became known as the *multi/meta-property* extension. [Blueprints]

There have been many efforts in the graph computing space to create a standard graph query language. These efforts have been advocated by standards committees, vendors, and educational institutions with varying degrees of success, but never universal acceptance. While TinkerPop has never promoted Gremlin as a standard, it has perhaps seen the most widespread adoption across the various open source and commercial data system vendors. Even with Gremlin poised to claim itself to be a "standard," TinkerPop eschews this title as one language is not sufficient to express all means of graph data processing. Much like there is no universal programming language, there are deleterious gains to be had from a single universal query language. The desire to renounce Gremlin's rising status as a query language standard has been captured by the colloquial phrase: "If you see the Gremlin, kill the Gremlin." Furthermore, by promoting Gremlin bytecode over the Gremlin language, the Gremlin language sits side-by-side other graph languages: exalted by some, simply useful to others, and found unseemly to yet more.

For TinkerPop, a graph traversal machine bytecode is the more powerful concept because it is the foundational assembly language leveraged by all Gremlin language variants (i.e. Gremlin hosted in another programming language) and distinct languages (i.e. a query language that compiles to Gremlin bytecode). This means that any query language can have a compiler to Gremlin bytecode for execution against any TinkerPop-enabled data system. This generality allows the vendor (and its users) to promote whichever language(s) they deem fit for their particular use cases. TinkerPop is thus enriched by a diverse collection of query languages that account for the many ways in which users think about and interact with graphs. The fable of the "conceptual snare" was the inspiration for this direction and will be further discussed in §III.

TinkerPop3 was the first version of the project that made its virtual machine architecture explicit. The relationship between the terminology used by TinkerPop3 and that espoused by standard virtual machine architectures are presented in the Table I [3].

TinkerPop3	**Virtual Machine**
Gremlin language	Programming language
Gremlin bytecode	Bytecode
Gremlin traversal	Machine code
Data storage system	Machine memory
Data processing engine	Machine CPU

TABLE I: TinkerPop and virtual machine terminology

The artwork created during the TinkerPop3 era took a new direction. Drawings emerged that showed many Gremlins interacting with one another as if Gremlin was no longer a single individual, but a collection of individuals trying to solve the problem of existence via their aggregate behavior through the graph. Gremlin was less the language and more generally a traversal, where real-world traversals can birth an unfathomable number of traversers at execution time (sometimes beyond the 64-bit long space). "Multiplicity" became a general theme of TinkerPop3, where agnosticism reigned in the many Gremlin language variants, the many distinct query languages, the many data storage systems, and even, for a moment, but without the finishing touch, the many data processing engines. This was made possible because of the virtual machine aspects of the project and, most importantly, by its bytecode specification that promoted universal interoperability.

III. BEHIND THE VEIL WITH TINKERPOP4: THE TRAVERSER

TinkerPop4 has been in design since the beginning of 2018 and will start development in the Spring 2019. At the tail end of TinkerPop3's creative development cycle, the stories began to ignore common graph use cases and instead, focused on problems in graph computing theory [16]. Two particular tales have played an important role in the design of TinkerPop4 [19, 20]. In [19], Gremlin decides that endlessly searching the graph will only yield more knowledge, not a deeper, truer, more virtuously astute awareness of reality. He reckons that the best way to understand The TinkerPop is to build a TinkerPop – or at least, what he believes the TinkerPop to be. Gremlin constructs a traversal machine within the graph structure G complete with Ψ-based traversal strategies and a Ψ-based execution engine. The reader gains a proof of the Turing Complete nature of Gremlin via the demonstration of a recursive universal traversal machine, but Gremlin, unfortunately, suffers the inefficiencies of virtualization and the self-reflective [8] consequences of having both program and data in the same vertex/edge address space of G. While Gremlin had created a universal machine within The TinkerPop, he still was unable to make direct reference to *The* TinkerPop – the simulation is not the reality. Gremlin became plagued by an existential crises which was only deepened with a question posed by some unknown being within The TinkerPop: "Is it possible for software to understand the hardware that is executing it? From the gates, to the electrons, to the physics, to the what? Where is the bottom?"

Gremlin's desire to see "behind the veil" was inspired by the Indo-Aryan concept of māyā and the grays of modern psychoanalysis [14, 27]. In these reports, there exists a more fundamental layer than that espoused by modern physics and the layers above are controlled by sophisticated beings who are executing the "reality" that mankind is aware of. When man is able to see "behind the veil," man is no longer the means of their computation, but the computation itself. Gremlin gives credence to this metaphysics and decides that if it is true, he must see it for himself.

> A true metaphysics must be a theory of experience, and not a guesswork as to what is at the back of it hiding itself under a veil. [12]

The downward decent through the layers of abstraction from language, to bytecode, to traversal, to a single traverser $t \in T$ had transformed the fun loving Gremlin of bygone versions into an ascetic monk attempting, at all costs, to see (and be) The TinkerPop. Perhaps The TinkerPop was just a thought in his head whose only substance is the endless, barren search he called a life. In [20], a tired and defeated Gremlin simply lets go – "If you see the Gremlin, kill the Gremlin."

Since the inception of the TinkerPop project, there has been little discussion of what The TinkerPop actually *is* from a computational perspective. The mythology of The TinkerPop has always remained vague with amorphous parables hinting that it is the engine of existence. TinkerPop4 will pull attention away from Gremlin and focus on the development of the lore and specification of The TinkerPop as *the* universal graph traversal machine. Through all his efforts, Gremlin only had to realize that he himself was The TinkerPop, albeit a fractal subset of a complex process shared with millions, billions, trillions, quadrillions, quintillions, … of others traversers [23]. The TinkerPop is simply $\forall t \in T$. The machine is the traverser(s). Together each t forms a fragment of The TinkerPop and reality is Gremlin himself reflected as far as he wants to shine. This mythos will require a new terminology which will render Gremlin simply as "the language" (his conceptual snare) and "Tinker-" will be used as the prefix for "the machine" (see Table II). The graph, the traversals, the languages, the bytecode, the strategies, and everything else are just aspects of his machine friends helping him to be all the t that he can be. When these pieces are combined, the act of graph computing is The Tinker'Pop'.

TinkerPop3	**TinkerPop4**	**Description**
Gremlin language	Gremlin language	Language spec.
Gremlin variant	Gremlin variant	Hosted impl.
Distinct language	Distinct language	Non-Gremlin lang.
Variants+distincts	TinkerLanguage	Any language
Gremlin bytecode	TinkerCode	β language
Language provider	Language provider	β compiler
Structure provider	Structure provider	G storage
Process provider	Process provider	Ψ processing
Gremlin machine	TinkerMachine	$\beta/G/\Psi$ integrator

TABLE II: TinkerPop Terminology

The TinkerMachine will seamlessly unify three types of software: data query languages, data storage systems, and data processing engines. TinkerPop1 and TinkerPop2 have always been agnostic towards the underlying data storage system. This has allowed any structure provider to leverage TinkerPop technology. TinkerPop3 was agnostic to the data query language with the introduction of a Turing Complete bytecode specification. This has allowed any language provider (graph or otherwise) to leverage any TinkerPop-enabled data system. Now with TinkerPop4, a new component will be plug-and-playable: the data processing engine. This will enable any data processor (e.g. iterative systems, message passing systems, map/reduce systems, scatter/gather systems, reactive systems, etc.) to move traversers about any TinkerPop-enabled data system. Thus, any query language can be used to control any processing engine to analyze graphs in any storage system. TinkerPop4 will also provide an architecture for developing a TinkerMachine in any programming language. This will make it so that the TinkerMachine is not constrained to a Java implementation, but can be developed in other "data system languages" such as C/C++, Go, Erlang, etc. and all TinkerMachine implementations will execute in the same manner, leveraging a standardized bytecode language. In summary, the three components of the TinkerMachine that can be integrated from 3rd party sources are itemized below.

- **Data query language**: TinkerMachine will support any query/programming language that can compile to bytecode β (see §III A).
- **Data storage system**: TinkerMachine will support any data storage system whose data can be interpreted as a property graph G (see §III B).
- **Data processing engine**: TinkerMachine will support any data processing engine which can migrate traversers according to Ψ (see §III C).

TinkerPop's goal is to accelerate the advances made in the graph computing space by leveraging the advances made across the entire computing field. Some teams excel at language development (specifications, compilers, verification), some teams excel at data storage (distribution, failure handling, consistency), and finally some teams excel at data processing (load balancing, thread management, serialization). Apache TinkerPop capitalizes on what each team does best by enabling the seamless unification of their best efforts into high-end graph computing systems.

The following diagram details all the major components of TinkerPop4. The remaining subsections will discuss the particulars of each subsystem and their role in The Tinker'Pop'.

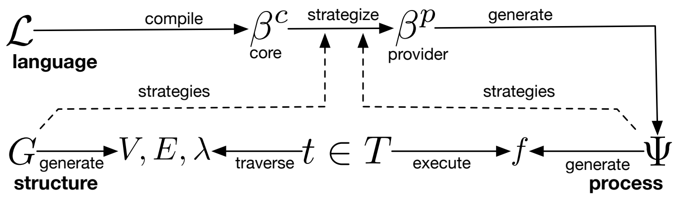

FIG. 1: TinkerPop4 System Architecture

A. Data Query Langauge

The Gremlin language will have a host language agnostic specification called "Gremlin." It will specify step names, step signatures, constants, and the rules for the composition of steps. Any implementation of that specification within a host programming language will be called a "Gremlin language variant." Thus, Gremlin-Java, which in TinkerPop3 was the specification, will be on equal footing along side Gremlin-Python, Gremlin-.NET, Gremlin-JavaScript, etc. Next, any query language that maintains a TinkerCode compiler (e.g. SPARQL [11, 28], SQL [2], Graql, etc.) will be called a "distinct language." All Gremlin language variants and distinct languages are under the general category of TinkerLanguages.

TinkerPop4 will extend the bytecode and compiler advances made in TinkerPop3. One of the major changes will be two sets of instructions: core instructions (β^c) and provider instructions (β^p). Both are under the general category of TinkerCode. Core bytecode instructions will form a reduced, Turing Complete instruction set that all languages will translate to. This is the machine independent assembly language. However, during bytecode strategy application, both structure and process providers will be able to insert custom, namespaced provider instructions of the same $[op, arg^*]$ form. These instructions specify custom Ψ-steps which are the integration points between the data storage system and data processing engine. If $\mathcal{L}$ is a TinkerLanguage, then the full translation process from language to traversal is

$$\mathcal{L} \longrightarrow_{\text{compile}} [\, \beta^c \longrightarrow_{\text{strategize}} \beta^p \longrightarrow_{\text{generate}} \Psi \,],$$

where the TinkerMachine's role lies within the brackets. Note that the TinkerMachine is not responsible for the compilation of a TinkerLanguage to TinkerCode. That is the sole responsibility of the language provider. An instance of a language-to-traversal translation process for an example data storage system that supports global indices is provided below.

$$\begin{array}{ll} \texttt{g.V().has('name','vadas')} & \in \mathcal{L} \\ \texttt{V[], has[name,vadas]} & \in \beta^c \\ \texttt{ex:vertex-idx[name,vadas]} & \in \beta^p \\ \texttt{ExVertexIndexStep(name,vadas)} & \in \Psi. \end{array}$$

Finally, because both TinkerPop3's and TinkerPop4's bytecode specifications are Turing Complete and have a similar $[op, arg^*]^*$ structure, there will be an effort to create a bi-directional translator between the two specifications so that aspects of TinkerPop3 can work with aspects of TinkerPop4 and vice versa.

In TinkerPop3, there were two ways of interacting with a TinkerPop-enabled system: 1.) direct graph objects or 2.) GremlinServer. In §III B, the removal of direct graph objects will be discussed. In terms of GremlinServer, the user was allowed to submit bytecode or a "script" written in one of the many script engine languages supported by the JVM. The goal of TinkerPop4 is to ensure that the TinkerMachine is agnostic to the implementing programming language and thus, there will be no ability to submit scripts to the TinkerMachine. Serialized bytecode will be submitted, the query executed, and a serialized stream of results will be returned. This will greatly simplify the I/O requirements of any TinkerMachine implementation.

The success of any software endeavor rests on the test suite that verifies its intended semantics. This is all the more important in a project such as TinkerPop where technologies from various diverse providers must come together and interact in concert at the TinkerMachine. TinkerPop1 and TinkerPop2 provided a test suite to ensure that that the underlying data storage system maintained a proper property graph implementation. In TinkerPop3, the test suite was extended to make sure all Gremlin language variants and distinct query languages correctly answered a standard suite of queries. TinkerPop4 will drop the G-specific test suite and focus solely on a language-based test suite as it is ultimately the language that drives the TinkerMachine and semantic certainty will be approached by testing every combination of language, storage system, and processing engine.

A review of the major changes at the data query language level are itemized below.

- **A Gremlin language variant specification**: A host language agnostic Gremlin language specification will be published for use by Gremlin language variant providers. This document will define step signatures and their rules for composition.
- **A generalized parameterization**: Leveraging runtime results in traversal step parameters was available primarily in the mutation steps (e.g. `addV`, `addE`, etc.). Dynamic parameterization will be generalized to all steps such that, for example, `out(x)` will be dynamically determined by what the current traverser references as `x`.
- **An extended step library**: There is a growing need for string manipulation and complex math functions in the Gremlin language. The Gremlin language will be extended to handle these new requirements.
- **An attempt to remove lambda support**: Lambdas were a critical element of the Gremlin language prior to TinkerPop3, but in that version the expressiveness of the language improved such that reliance on lambdas reduced considerably. TinkerPop4 will examine the possibility of Gremlin without any lambda support, which will improve security and simplify the codebase.
- **A more concise bytecode instruction set**: The number of core operands will be reduced to a Turing Complete set denoted β^c.
- **A custom bytecode infrastructure**: Structure and process providers will be able to insert custom β^p instructions to take unique advantage of their system-specific optimizations.
- **A bytecode optimizer**: Traversal strategies will be abandoned in favor of bytecode strategies which will operate at the bytecode level, not at the traversal/machine code level. Simpler regular expression pattern matching and transformations will be possible over a recursive list of $[op, arg^*]^*$ instructions versus over a nested, machine-specific traversal object.

- **An extended optimization algebra**: A number of traversal strategies were not implemented in TinkerPop3 because they were difficult to reason on at the traversal level. With all optimization occurring at the bytecode level in TinkerPop4, a more complete path algebra will emerge.

- **A generalized runtime optimizer**: The `match`-step based runtime optimizer will be generalized to support single patterns that can be explored by multiple algebraically equivalent patterns and thus, the pattern that executes fastest and/or with less memory will be leveraged in the final computation.

- **A deprecation of bytecode-level step-modulators**: There are various inconsistencies in the bytecode of TinkerPop3 that will be rectified in TinkerPop4. In particular, Gremlin step-modulators such as `by()`, `as()`, `to()`, `from()`, etc. will act at the Gremlin language variant's compiler level as "pop-push" bytecode stack operators affecting the previous instruction and will not be represented as instructions at the TinkerCode level.

- **A language driven test suite**: TinkerPop4 will verify that all compiling languages answer a standard suite of queries correctly where, in turn, those answers will require the correct semantics in both the underlying tested storage system and processing engine.

- **A bytecode/result serializer and protocol**: The only means of communicating with the TinkerMachine is via bytecode and a compact, efficient, platform-independent serialization format will be created. There will no longer be direct access to the underlying data storage system's graph objects. All communication will be via a language agnostic network protocol for sending bytecode and receiving results.

- **A bytecode translator**: A translator will be developed that maps TinkerPop3's Gremlin bytecode to TinkerPop4's TinkerCode. This will smooth user adoption and enable aspects of TinkerPop3 to leverage aspects of TinkerPop4 and vice versa.

B. Data Storage System

A structure provider is any developer of a data storage system. When the provider implements custom bytecode instructions in β^p and subsequent traversal steps in Ψ, this exposes their underlying data structure to be interpreted as a property graph G. At this point, the provider's system is considered TinkerPop-enabled. If G is the underlying graph storage system, then

$$G \longrightarrow_{\text{generate}} [\ V, E, \lambda \longleftarrow_{\text{traverse}} t \in T\ ]\ ,$$

where the TinkerMachine's role lies in ensuring the traversers in T access the vertex, edge, and property data in G as dictated by Ψ.

A significant change to come is the definition of the property graph G. Originally, in TinkerPop1 and TinkerPop2, the property graph specification was inspired by the Neo4j graph database, where every vertex and edge has a set of key/value pair "properties" associated with them. In later years, use cases emerged that required a vertex to support multiple property values for the same key. Moreover, it was important that these "multi-properties" had properties on them called "meta-properties." The uses cases that spawned the inclusion of multi- and meta-properties in TinkerPop3 were audit, provenance, permission, and time-encoded graphs. Unfortunately, the implementation of multi- and meta-properties in TinkerPop3 is semantically "clunky" and as such TinkerPop4 plans to rectify the situation by leveraging the RDF literal model [9]. In TinkerPop4, edges will be able to point to both vertices and literals, where the set of literals $\mathbb{L} = \Sigma^* \cup \mathbb{N} \cup \ldots \cup \mathbb{R}$. A simple isomorphism exists between TinkerPop3's and TinkerPop4's property graph definition as vertex properties will be literals and meta-properties will be properties on those edges leading to literals [6]. Thus, TinkerPop4's property graph $G = (V, E, \lambda)$, where $E \subseteq V \times \Sigma^* \times (V \cup \mathbb{L})$ and $\lambda : E \times \Sigma^* \rightarrow \mathbb{L}$. This will yield a graph that looks similar to the RDF graph specification save that edges, and only edges, can have any number of key/value pairs associated with them. The relationship between multi/meta-properties and literals is presented in Table III.

TinkerPop3	**TinkerPop4**
`out().values()`	`out().literals()`
`out().properties().value()`	`out().literals()`
`out().properties()`	`out().literalE()`
`out().values(‘name’)`	`out().literals(‘name’)`
`out().properties(‘name’).value()`	`out().literals(‘name’)`
`out().valueMap()`	`out().literalMap()`
`properties(‘name’).valueMap()`	`literalE(‘name’).valueMap()`
`properties(‘name’).values()`	`literalE(‘name’).values()`
`properties(‘name’).values(‘acl’)`	`literalE(‘name’).values(‘acl’)`
N/A	`literalE(‘name’).inL()`
`properties().properties()`	`literalE().properties()`
`outE().properties()`	`outE().properties()`
`outE().valueMap()`	`outE().valueMap()`
`outE().values(‘time’)`	`outE().values(‘time’)`

TABLE III: Gremlin4 and Literal Vertices

Another important change will be the removal of the `structure/` interfaces that define `Graph`, `Vertex`, `Edge`, `Property`, etc. Users will no longer have access to the underlying storage system via direct objects. Instead, all reads and writes will be through a TinkerLanguage and thus, via the TinkerMachine. Every structure provider will develop a set of provider instructions in β^p and consequently, a set of custom traversal steps in Ψ that will return unconnected TinkerPop-specific graph objects. TinkerPop4 will have no steps that access data from the data storage system. Those steps will be fully developed by the provider. The semantics of the provider's property graph encoding will take place at the language level where a suite of test queries will ensure that the provider has a correctly encoded graph.

A review of the major changes at the data storage system level are itemized below.

- **No vertex properties or meta-properties**: The only elements that will support key/value properties will be edges. A new type of literal vertex will be introduced which will make G analogous to an "RDF property graph." This structure will support the concept of multi- and meta-properties more elegantly, though they will not be called multi- and meta-properties.
- **No graph interface**: The graph `structure/` interfaces will no longer exist. It will be up to particular provider/vendor step implementations to handle populating vertex, edge, property, etc. objects that are universal to all TinkerPop-enabled systems. These graph objects will not be connected and will only provide the associated incident data (e.g. label, id).
- **No graph test suite**: There will no longer be a graph test suite. All testing of the semantics of the underlying storage system will be via the language test suite discussed in §III A.
- **No transaction interface**: Transactions will be handled by the graph system and will not be wrapped by a TinkerPop interface package. This is in line with the trend in TinkerPop3 to remove index and schema handling support as such subsystems are too varied among providers to be elegantly generalized.

C. Data Processing Engine

TinkerPop4's traversal processing framework will focus on the agency of a traverser and the logical entailments of that agency. In TinkerPop3, traversals were the primary thread of execution, where the parallel distributed OLAP graph computer engine had to isolate each step to ensure functional, thread safe semantics. TinkerPop4 plans to support various processing models including single-machine serial, single-machine parallel, multi-machine distributed, and all the various distributed computing techniques and engines in existence that implement them. As such, it is important to develop a framework that is thread-safe and distributed at its core. In order to do this, TinkerPop4 will provide a *traverser-centric* framework where the primary thread of execution lies at the individual traverser and thus, each traverser can evaluate independent of all other traversers in the traversal swarm. One of the major consequences of this effort will be the co-location of a traverser with the vertex/edge it is referencing and thus, TinkerPop4 will support dynamic query routing. There were preliminary efforts to accomplish this in TinkerPop3 with the "graph actors" framework, but this was abandoned in lieu of designs for TinkerPop4.

If f is the function of a step in Ψ, then the execution of a traverser $t \in T$ will appear as

$$[\, t \in T \ \longrightarrow_{\text{execute}} f] \longleftarrow_{\text{generate}} \Psi,$$

where $f : T^* \rightarrow T^*$ is the standard traverser stream construct employed by every step of a traversal. A review of the major changes at the data process engine level are itemized below.

- **A traverser-centric framework**: In TinkerPop3, traversers were processed by traversals. In TinkerPop4, traversers will be the primary agent/thread of execution and will be responsible for the evaluation of the step functions. Traversers will be decoupled, stateful entities moving about G according to Ψ and can execute independently of one another. This will more easily enable the leveraging of various serial, parallel, and distributed processing techniques.
- **A traverser-stream function library**: Traversal steps will be decoupled from their executing function. It will be up the processing engine to determine how to compose steps from a `map`, `flatMap`, `filter`, etc. library of functions.
- **A completely stateless traversal**: All step state will be completely removed. Side-effect steps will store their side-effects in "stationary", side-effect traversers. This will ease thread safety concerns when integrating with arbitrary process providers.
- **A configurable traverser bulk**: Traversers will be able to support any standard numeric object and split/merge function as their *bulk* [18]. This will enable massive-scale traversals that spawn localized populations larger than can be counted in the 64-bit long space and, more interestingly, will enable wave-based graph computing with complex numbers in $\mathbb{C}$ [24].
- **A graph partition interface**: Data storage systems will be able to specify where a particular vertex/edge/etc. is located in the cluster. This will enable the co-location of process (traverser) and structure (vertex/edge/etc.). An interesting entailment of this feature will be query routing.

IV. CONCLUSION

The TinkerPop project has been through many changes over the years. The one constant is that it has always strived to provide a vendor-agnostic graph computing framework. In the beginning, TinkerPop provided an open graph framework for integration with any data storage system. More recently, TinkerPop has opened up its language framework for any data query language to leverage. Moving forward, TinkerPop will open up its traversal framework to enable the integration of any data processing engine. While the common conception of TinkerPop is that it is about the Gremlin language, the mythology of TinkerPop tells a different story. The fables of The TinkerPop have taken the project to realize the Gremlin language as a particular style of expressing a graph computation and if Gremlin is to understand The TinkerPop, he must shed this skin and look for a deeper meaning in himself, t. TinkerPop will continue to downplay the importance of the Gremlin language in support of a bytecode-based graph traversal machine that is open to attract the work of experts in the fields of data storage, data processing, and query language design.

Acknowledgments

This article represents the authors' current boundaries of thought after many years of exploring the wonderful world of graphs. Many individuals have supported the Apache TinkerPop$^{\text{TM}}$ project over the years from project members to individuals on related efforts in both the open source and commercial sectors. Particular individuals who have provided a steadfast, concerted effort on TinkerPop over the years include Daniel Kuppitz (Gremlin language design) and Ketrina Yim (Gremlin artwork). Finally, the authors would like to thank Peter Neubauer and Joshua Shinavier for their original collaborations with Marko A. Rodriguez in the founding of the TinkerPop project back in 2009. The authors look forward to the developments to come in "TinkerPop4: Rise of Enki."

[1] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. A budget-based algorithm for efficient subgraph matching on huge networks. In *Workshops Proceedings of the 27th International Conference on Data Engineering*, pages 94–99, 2011. doi: 10.1109/ICDEW.2011.5767618. URL `http://dx.doi.org/10.1109/ICDEW.2011.5767618`.

[2] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. ISSN 0001-0782. doi: 10.1145/362384.362685.

[3] I. D. Craig. *Virtual Machines*. Springer, 2005.

[4] K. Dharma. *Mahabharata*. Om Books International, 1999. ISBN 978-81-87108-99-3.

[5] H. Griffith. The gremlin question. *Royal Airforce Journal*, (13), April 1942.

[6] O. Hartig. Reconciliation of RDF* and property graphs. Technical report, University of Waterloo, 2014. URL `http://arxiv.org/abs/1409.3288`.

[7] G. Lachman. *The Secret Teachers of the Western World*. TarcherPerigee, December 2015. ISBN 0399166807.

[8] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, January 1987.

[9] E. Miller. An introduction to the Resource Description Framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, November 1998. doi: 10.1002/bult.105.

[10] R. K. Narayan. *The Ramayana*. Penguin Group, 1972.

[11] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical report, World Wide Web Consortium, October 2004. URL `http://www.w3.org/TR/rdf-sparql-query/`.

[12] S. Radhakrishnan. *Indian Philosophy: Volume 1*. Oxford University Press, 1923.

[13] M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2: 1019–1025, 1999.

[14] M. A. Rodriguez. A methodology for studying various interpretations of the N,N-dimethyltryptamine-induced alternate reality. *Journal of Scientific Exploration*, 21 (1):67–84, 2007. ISSN 0892-3310.

[15] M. A. Rodriguez. Grammar-based random walkers in semantic networks. *Knowledge-Based Systems*, 21(7): 727–739, 2008. doi: 10.1016/j.knosys.2008.03.030. URL `http://arxiv.org/abs/0803.4355`.

[16] M. A. Rodriguez. *Emergent Web Intelligence: Advanced Semantic Technologies*, chapter General-Purpose Computing on a Semantic Network Substrate, pages 57–104. Advanced Information and Knowledge Processing. Springer-Verlag, June 2010. ISBN 78-1-84996-076-2. URL `http://arxiv.org/abs/0704.3395`.

[17] M. A. Rodriguez. A TinkerPop story, December 2011. URL `https://markorodriguez.com/2011/12/12/a-tinkerpop-story/`.

[18] M. A. Rodriguez. The Gremlin graph traversal machine and language. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10. ACM, October 2015. ISBN 978-1-4503-3902-5. doi: 10.1145/2815072.2815073. URL `http://arxiv.org/abs/1508.03843`.

[19] M. A. Rodriguez. A Gremlin implementation of the Gremlin traversal machine, October 2016. URL `https://www.datastax.com/dev/blog/a-gremlin-implementation-of-the-gremlin...`

[20] M. A. Rodriguez. The Von Gremlin architecture, August 2017. URL `https://www.datastax.com/dev/blog/the-von-gremlin-architecture`.

[21] M. A. Rodriguez and P. Neubauer. A path algebra for multi-relational graphs. In *Proceedings of the $2^n d$ International Workshop on Graph Data Management*, pages 128–131, Hannover, Germany, April 2011. IEEE. doi: 10.1109/ICDEW.2011.5767613. URL `http://arxiv.org/abs/1011.0390`.

[22] M. A. Rodriguez and J. Shinavier. Exposing multi-relational networks to single-relational network analysis algorithms. *Journal of Informetrics*, 4(1):29–41, 2009. doi: 10.1016/j.joi.2009.06.004. URL `http://arxiv.org/abs/0806.2274`.

[23] M. A. Rodriguez and D. J. Steinbock. A social network for societal-scale decision-making systems. In *Pro-*

ceedingss of the North American Association for Computational Social and Organizational Science Conference, Pittsburgh, PA, 2004. URL `http://arxiv.org/abs/cs.CY/0412047`.

[24] M. A. Rodriguez and J. H. Watkins. Quantum walks with Gremlin. In L. Bender, editor, *Proceedings of the GraphDay Conference*, volume 1, pages 1–16, Austin, Texas, January 2016. URL `https://arxiv.org/abs/1511.06278`.

[25] M. A. Rodriguez, D. Kuppitz, and K. Yim. Tales from the TinkerPop, July 2015. URL `https://www.datastax.com/dev/blog/tales-from-the-tinkerpop`.

[26] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, July 1993. ISBN 0262181231.

[27] R. J. Strassman. *DMT the Spirit Molecule: A doctor's revolutionary research into the biology of near-death and mystical experiences*. Park Street Press, Rochester, VT, 2001.

[28] H. Thakkar, D. Punjani, J. Lehmann, and S. Auer. Two for one: querying property graph databases using SPARQL via Gremlinator. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 12:1–12:5, Huston, TX, June 2018. ACM. ISBN 978-1-4503-5695-4. doi: 10.1145/3210259.3210271.

[29] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1937.